Capítulo 5: Arrays

Arrays

- **Visão Geral**
- **Construindo arrays**
- **Métodos que modificam o array original (mutator methods)**
- **Métodos que NÃO modificam o array original (accessor methods)**
- **Métodos que iteram no array (iterator methods)**
- **Programação declarativa**
  - foreach
  - map
  - filter
  - reduce

Arrays: visão geral

São dinâmicos em Javascript

Ao contrário de muitas linguagens, onde o array tem tamanho fixo:

Ex: const meuArray=[]; meuArray.push("oi");

São heterogêneos (um único array aceita diferentes tipos de dados)

Na prática, porém, arrays são homogêneos

Indexados a partir do zero

Acessar elementos de índices inexistentes não gera erro

Retorna *undefined*

Arrays apresentam a propriedade *length,* que retorna o número de elementos da matriz.

Muitas linguagens de programação oferecem suporte a arrays com índices não numéricos.

Os arrays com índices nomeados são chamadas de arrays associativos ou hashes (mas já vimos isso com outro nome, não?)

JavaScript não oferece suporte a arrays com índices nomeados. Em JavaScript, os arrays sempre usam índices numerados .

Construindo arrays

1) Usando a função construtora Array:

```
let nomes=new Array(“José”, “Maria”, “João”);
console.log(nomes[0]);//”José”
nomes[2]=”Pedro”  //[ "José", "Maria", “Pedro”]
nomes[3]=”Bia” //[ "José", "Maria", “Pedro”,”Bia”]
```

2) Usando a notação literal:

```
let nomes=[ "José", "João", "Maria"]
nomes[8]=”Ana”;
console.log(nomes);
//["José", "João", "Maria", empty × 5, "Ana"]
```

Métodos que alteram o array original (mutator methods)

Em Javascript, alguns métodos de arrays modificam a estrutura original do array (*mutator methods*)e outros não (*accessor methods*). Veremos os primeiros agora.

push(elemento)

Insere elemento ao final do array (e atualiza length)

pop()

Remove último elemento do array e atualiza length

sort()

Ordena elementos do array

Se *funçãoDeComparação* for omitida, o array será ordenado de acordo com a pontuação de código do Unicode (code-point)

Considere o array original:

```
let nomes=[“José”, “Maria”, “João”];


nomes.push(“Bia”);
 //[“José”, “Maria”, “João”, “Bia”]
nomes.pop()
//["José", "Maria"]
nomes.sort();
// ["José", "João", "Maria"];
//notar que no Unicode ‘ã’ vem depois de ‘s’
//assim como números vêm antes de maiúsculas, que vêm antes de minúsculas, etc.
```

Métodos que alteram o array original (mutator methods)

delete

Operador que apaga elemento, sem reposicionar os demais

Posição deletada contem *undefined*

shift()

Remove primeiro elemento do array, reposiciona elementos e atualiza length

unshift(elemento)

Adiciona elemento no início do array e atualiza length

Continue com o array original:

```
let nomes=[“José”, “Maria”, “João”];

delete nomes[0];
// [empty, "Maria", “João”];

nomes.shift();
//[“Maria”, “João”]

nomes.unshift(“Bia”)
//["Bia", "Maria", "João"]
```

Métodos que alteram o array original: splice

array.splice(indice[, deleteCount[, elemento1[, ...[, elementoN]]])

índice: Índice o qual deve iniciar a alterar a lista. Se maior que o tamanho total da mesma, nenhum elemento será alterado. Se negativo, irá iniciar a partir daquele número de elementos a partir do fim.

deleteCount: Um inteiro indicando o número de antigos elementos que devem ser removidos.

Se o parâmetro deleteCount não for especificado, todos os elementos até o fim da lista serão deletados.(ou seja, apenas um parâmetro)

Se deleteCount é 0, nenhum elemento é removido. Neste caso você deve especificar pelo menos um novo elemento.

elemento1, ..., elementoN: Os elementos a adicionar na lista. Se você não especificar nenhum elemento, splice simplesmente removerá elementos da mesma.

splice() retorna novo array com elementos removidos

Não confundir com o método *slice()*, que NÃO modifica o array, como veremos adiante

O método splice() altera o conteúdo de uma lista, adicionando novos elementos enquanto remove elementos antigos:

let nomes=["José", "João", "Maria", "Bia"]

a.remove 0 elementos a partir do índice 2, e insere "Téo"

```
let removidos=nomes.splice(2,0,"Téo")
//nomes= ["José", "João", "Téo", "Maria", "Bia"]
//removidos=[]
```

b.remove 1 elemento do índice 3

```
removidos = nomes.splice(3,1)
//nomes=["José", "João", "Maria"]
//removidos=["Bia"]
```

Métodos que alteram o array original: splice

c. remove 1 elemento a partir do índice 2, e insere "Lara":

```
removidos=nomes.splice(2,1,"Lara")
//nomes= ["José", "João", "Lara", "Bia"]
//removidos=["Maria"]
```

d. remove 2 elementos a partir do índice 0, e insere "Luiz", "Victor" e "Gabriel"

```
removidos = nomes.splice(0,2,"Luiz","Victor","Gabriel")
//nomes=["Luiz", "Victor", "Gabriel", "Maria", "Bia"]
//removidos=["José", "João"]
```

Conclusões:

a) se pop() e shift() retiram elementos do fim e do início de um array, respectivamente, splice() retira de uma posição qualquer, determinada pelo índice.

b) Do mesmo jeito, push() e unshift() inserem elementos no fim e inicio do array e splice() insere em qualquer posição (considerando que elementos foram passados a partir do terceiro parâmetro)

Métodos que alteram o array original: splice (Exercícios)

Exercícios:

Seja o array:

let nomes=["José", "João", "Maria", "Bia"]

a) use o método splice para substituir “João” por “Téo”

b) Adicione a “Ana” ao final do array de duas maneiras diferentes.

c) Quais os métodos usados no item b?

Retornando ao sort: sort(funçãoDeComparação)

A *funçãoDeComparação* fornecida a sort() deve ter o seguinte formato:

```
function comparar(a, b) {
  if (a é menor que b em algum critério de ordenação) {
    return -1;
  }
  if (a é maior que b em algum critério de ordenação) {
    return 1;
  }
  return 0;   // a deve ser igual a b
}
```

Ex: Para comparar números ao invés de texto, a função de comparação pode simplesmente subtrair b de a. A função abaixo irá ordenar o array em ordem crescente:

```
function compararNumeros(a, b) {
    return a - b;
}
```

Ex:

```
var numbers = [4, 2, 5, 1, 3];
const compararNumeros=function(a, b) {
  return a - b;
};
console.log(numbers.sort(compararNumeros));
```

sort(funçãoDeComparação): exercício resolvido

Dado o seguinte array, ordene-o de acordo com o valor da chave *nome:*

```
var alunos = [
 { nome: 'Eduardo', idade: 21 },
 { nome: 'Maria', idade: 37 },
 { nome: 'André', idade: 45 },
 { nome: 'Téo', idade: 12 },
 { nome: 'Bia', idade: 67 },
 { nome: 'Zé', idade: 37 }
];
```

Resposta:

```
alunos.sort(function (a, b) {
 if (a.nome > b.nome) {
  return 1;
 }
 if (a.nome < b.nome) {
  return -1;
 }
  return 0;
});
```

sort(funçãoDeComparação): exercício para fazer

Dado o seguinte array, ordene-o de acordo com a *idade* de cada aluno:

Resposta:

```
var alunos = [
 { nome: 'Eduardo', idade: 21 },
 { nome: 'Maria', idade: 27 },
 { nome: 'André', idade: 25 },
 { nome: 'Téo', idade: 12 },
 { nome: 'Bia', idade: 17 },
 { nome: 'Zé', idade: 27 }
];
```

sort(funçãoDeComparação): exercício para fazer

Como vimos, o método sort ordena strings comparando o code-point de cada caracter, da esquerda para a direita. E se quiséssemos mudar o critério de ordenação para o comprimento das strings (ou seja, uma string com 4 caracteres viria antes de uma string com 5 caracteres)?

Elabore uma função de comparação que, aplicada ao sort(), ordene o seguinte array de acordo com o comprimento das strings:

var nomes=[“Maria”,”Ana”,”Zé”, “Téo”]

Resposta:

Dica: tamanho= length

Array: métodos que NÃO afetam array original (accessor methods)

Os métodos a seguir não mexem no array original, retornando sempre um novo array.

slice(*começo*, *fim*)

Copia uma determinada parte de um array e retorna essa parte copiada como um novo array.

Ao contrário de splice(), *fim* também é um deslocamento

O elemento *fim* não é incluído!

```
let nomes=[ "Zé", "Téo", "Ana", "Bia"]

let aprovados=nomes.slice(1,2)
//notar que nomes não foi alterado
```

concat(elem1, elem2,….,elemN)

cria um novo array unindo todos os elementos passados como parâmetro, na ordem dada.

Elementos podem ser strings, outros arrays, etc.

```
var resultado=nomes.concat(“Maria”)
//resultado=[ "Zé", "Téo", "Ana", "Bia", “Maria”]
resultado=nomes.concat([1,2,3], “Léo”);
//resultado=[ "Zé", "Téo", "Ana", "Bia", 1,2,3, “Léo”]
```

Array: métodos que NÃO afetam array original (accessor methods)

let frutas=[“Uva”, “Maçã”, “Pera”, “Uva”]

join(separador)

Junta elementos de um array em uma string separando-os por *separador.* É um toString() exclusivo de array.

Sem separador, separa elementos por vírgula

```
var str=frutas.join(“-”) //”Uva-Maçã-Pera-Uva”
str=frutas.join(“”)      //”UvaMaçãPeraUva”
```

indexOf(elemento)

Retorna o índice da ***primeira*** ocorrência de um dado valor no array, ou -1 se o valor não estiver incluso no array

```
var pos=frutas.indexOf(“Uva”) //0
```

lastIndexOf(elemento)

Retorna o índice da ***última*** ocorrência de um dado valor no array, ou -1 se o valor não estiver incluso no array

```
var pos=frutas.lastIndexOf(“Uva”) //3
```

Exercícios

Dados os arrays abaixo, faça o que se pede:

let frutas=["uva","maçã","pera","uva"]

let nomes=["Zé", "Téo", "Ana", "Bia"]

1) qual o índice da “Ana”?

2) gere uma string com os elementos de frutas separando-os por ‘+’

3) concatene os dois arrays, *nomes* e *frutas*

Métodos que iteram no array (iterator methods)

Uma operação trivial em arrays é iterar por seus elementos (ou seja, visitar cada um de seus elementos, ordenadamente, e executar uma função com o elemento da vez)

Os métodos de iteração *não* afetam o array original

Assim, vários métodos de Array aceitam funções como argumento, que serão chamadas a cada novo elemento do array (*callbacks*). Os métodos que veremos a seguir todos aceitam uma *callback*.

A função callback, presente em todos os métodos a seguir é executada em cada elemento. Ela recebe três parametros*:

function callback(valor, indice, array)){...}

valor (obrigatório)

O elemento atual sendo processado na array.

indice (opcional)

O índice do elemento atual sendo processado na array.

array (opcional)

O array de origem.

***na prática, na maioria das vezes, callback faz uso apenas do primeiro parâmetro (que é obrigatório)**

Array Iterator methods: every, some, find, includes

every(*predicate*)

testa se todos os elementos do array passam pelo teste implementado pela função fornecida

Se um dado elemento não passar no teste, o método ***every*** imediatamente retorna ***false***.

Caso contrário, se a função *callback* retornar *true* para todos elementos, o método ***every*** retorna *true*

Lembre-se que ***predicate*** é uma callback que retorna um booleano (false ou true)

Ex:

```
function maiorQueDez(element, index, array) {
  return element >= 10;
}
[12, 5, 8, 130, 44].every(maiorQueDez);   // false
[12, 54, 18, 130, 44].every(maiorQueDez); // true
```

Certamente, podemos reescrever a função acima como uma arrow function:

```
[12, 5, 8, 130, 44].every(x=>x>=10);   // false
```

Array Iterator methods: every, some, find, includes

some(*predicate*)

testa se ao menos um dos elementos no array passa no teste implementado pela função atribuída e retorna um valor *true* ou *false*

find(*predicate*)

executa a função uma vez para cada elemento do array:

Se encontrar um elemento onde a função retorna um valor verdadeiro, *find*() retorna o valor desse elemento da matriz (e não verifica os valores restantes), caso contrário, retorna *undefined*

includes(*element*)

determina se um array contém um determinado elemento, retornando true ou false apropriadamente

Dados o array e callback abaixo:

```
var idades = [3, 10, 18, 20];
function checaSeAdulto(idade) {
    return idade >= 18;
}
```

some():

```
idades.some(checaSeAdulto); //true
```

find():

```
idades.find(checaSeAdulto); // 18
```

Includes():

```
idades.includes(10); //true
```

Array Iterator methods: map, filter, forEach,reduce

As funções map, filter, forEach e reduce são fundamentais na *programação declarativa* do Javascript

Programação declarativa diz *o que* deve ser feito enquanto a imperativa diz *como* deve ser feito (passo-a-passo)

Cada uma das operações abaixo vai aplicar a *callback* em cada elemento do array, ficando a diferença entre elas no tipo de retorno de cada uma:

map: retorna outro array

filter: retorna array filtrado (subset do original)

forEach: não retorna nada

reduce: retorna um “escalar”

Uma abordagem imperativa para selecionar elementos pares de um array:

```
var numeros = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
var pares = [];
for (var i = 0; i < numeros.length; i++) {
 if (numeros[i] % 2 == 0) {
  pares.push(numeros[i]);
 }
}
console.log(pares); // (6) [0, 2, 4, 6, 8, 10]
```

O mesmo problema resolvido declarativamente:

```
var pares=numeros.filter(x=>x%2==0)
```

Array Iterator methods: map, filter, forEach,reduce

map(callback)

O método map() cria uma nova matriz com os resultados da chamada de uma função para cada elemento da matriz.

filter(predicate)

O método filter() cria uma nova matriz com elementos de matriz que passam no teste.

Ex:

```
var origem = [45, 4, 9, 16, 25];
function duplica(valor) {
  return valor * 2;
}
var destino = origem.map(duplica);


function par(valor){
   return valor%2==0;
}
var destino = origem.filter(par);
```

Array Iterator methods: map, filter, forEach,reduce

forEach(callback)

O método forEach() executa uma dada função em cada elemento de um array.

Ex:

```
function imprimeElementos(element, index, array) {
    console.log("a[" + index + "] = " + element);
}
[2, 5, 9].forEach(imprimeElementos);
```

Array Iterator methods: map, filter, forEach,reduce

reduce(callback(acc,cur[,idx,src])[, valorInicial])

O método reduce() aceita dois parâmetros: uma função callback e o *valorInicial*. A função *callback* (fornecida por você) é executada para cada elemento do array, resultando num único valor de retorno.

A *callback* recebe quatro parâmetros:

- Acumulador (acc):
- Valor Atual (cur)
- Index Atual (idx)
- Array original (src)

O valor de retorno da callback é atribuída ao acumulador. O acumulador, com seu valor atualizado, é repassado para cada iteração subsequente pelo array, que por fim, se tornará o valor resultante, único, final

Ex:

```
const array1 = [1, 2, 3, 4];

const reducer = (acc, cur) => acc + cur;


// 1 + 2 + 3 + 4

console.log(array1.reduce(reducer));

// saída esperada: 10


// 5 + 1 + 2 + 3 + 4

console.log(array1.reduce(reducer, 5));

// saída esperada: 15
```

Exercícios

```
const tarefas = [
 {
  'nome'    : 'estudar Javascript',
  'duracao' : 150
 },
 {
  'nome'    : 'correr',
  'duracao' : 60
 },
 {
  'nome'    : 'netflix',
  'duracao' : 120
 }
];
```

1) Crie uma matriz com as tarefas que duram mais de 1 hora

2) Cria uma nova matriz onde as durações são dobradas

3) Imprima o nome e duração das tarefas ao lado

4) Calcule a soma das durações da matriz ao lado

5) implemente a questão 1 usando forEach